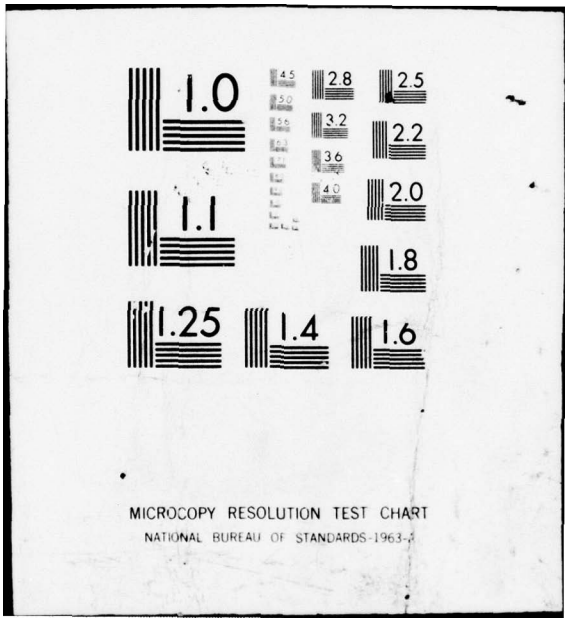
AD-A044 200 RAND CORP SANTA MONICA CALIF F/G 17/2
INTERPROCESS COMMUNICATION EXTENSIONS FOR THE UNIX OPERATING SY--ETC(U)
JUN 77 C A SUNSHINE F49620-77-C-0023
UNCLASSIFIED R-2064/1-AF NL

| OF |
AD
A044200

END
DATE
FILMED
10-77
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

R-2064/1-AF
June 1977

Interprocess Communication Extensions for the UNIX Operating System: I. Design Considerations

Carl Sunshine

A Project AIR FORCE report
prepared for the
United States Air Force

Rand
SANTA MONICA, CA. 90406

The research reported here was sponsored by the Directorate of Operational Requirements, Deputy Chief of Staff/Research and Development, Hq. USAF under Contract F49620-77-C-0023. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

Published by The Rand Corporation

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: R-2064/1-AF (14)	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Interprocess Communication Extensions for the UNIX Operating System: I. Design Considerations (6)	5. TYPE OF REPORT & PERIOD COVERED: (9) Interim report / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): (10) Carl A. Sunshine	8. CONTRACT OR GRANT NUMBER(s): (15) F49620-77-C-0023
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The Rand Corporation, 1700 Main Street, Santa Monica, Ca. 90406	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Project AIR FORCE Office (AF/RDQA), Directorate of Operational Requirements, Hq USAF, Washington, D.C. 20330	12. REPORT DATE: (11) June 1977 / 13. NUMBER OF PAGES: 31 (12) 36p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report): Approved for Public Release; Distribution Unlimited	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report): No restrictions	
18. SUPPLEMENTARY NOTES	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number): Data Processing; Data Transmission; Computer Systems Programs; Coding Theory; Software (computers); Operating Systems (Computers); Interprocess Communication	
20. ABSTRACT (Continue on reverse side if necessary and identify by block number): see reverse side	

296600 B

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The UNIX operating system for the PDP-11 series of minicomputers has gained wide popularity in academic and government circles. This report considers interprocess communication (IPC) facilities with the goal of developing an improved IPC capability for UNIX. An outline of the major issues involved in providing IPC is developed based on a survey of the literature, and UNIX IPC facilities are described in terms of this outline. By considering new applications being developed under UNIX, several shortcomings in the standard IPC facilities are identified, including the inability of "unrelated" processes to communicate, the inability to wait for multiple inputs, and primitive synchronization facilities. Techniques to provide desirable improvements are suggested, including named ports, message ports, improved signals, and message facilities. Ports appear to have the highest benefit/cost ratio, and their implementation is described in a companion report, R-2064/2-AF. (Author)

R-2064/1-AF
June 1977

Interprocess Communication Extensions for the UNIX Operating System: I. Design Considerations

Carl Sunshine

A Project AIR FORCE report
prepared for the
United States Air Force

Rand
SANTA MONICA, CA. 90406

blank

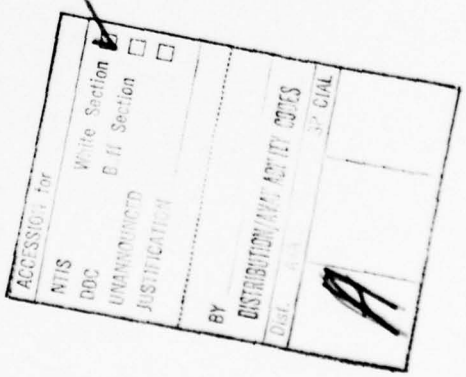
PREFACE

The UNIX operating system for the PDP-11 series of minicomputers has gained wide popularity in academic and government circles. Under the Project AIR FORCE (formerly Project RAND) study effort "Information Sciences Research," The Rand Corporation is engaged in analyzing, evaluating, and developing computer operating system concepts with UNIX. Recent work has dealt with such topics as security, file systems, performance, user interfaces, network access, and office automation.

This report, together with its companion report R-2064/2-AF,* describes the current state of work in the area of interprocess communication. A reasonable familiarity with UNIX is required to understand some detailed points, but sufficient background material is presented for the reader with a general knowledge of operating systems to follow the discussion. The report is aimed at computer system analysts and researchers concerned with operating systems supporting multiple processes, particularly for interactive applications.

---

* Steven Zucker, Interprocess Communication Extensions for the UNIX Operating System: II. Implementation, The Rand Corporation, R-2064/2-AF, June 1977.



SUMMARY

In developing new applications for the UNIX operating system, several shortcomings in the standard interprocess communication (IPC) capabilities have become apparent. To fully understand these problems and to develop effective solutions, a study of IPC techniques, including a survey of existing implementations, was undertaken. Cooperating processes within a computer operating system need to communicate for two main purposes: data transfer and synchronization. In general, port-like facilities are provided for transfer of streams of data, while semaphore or signal type primitives provide for synchronization. Message facilities share features of both data transfer and synchronization.

To provide a framework for discussion, Sec. II presents an outline of important features that any IPC system must include. The main points of this outline are connection establishment, resource allocation, functions available, receiver unblocking, and relation with other input/output. Section III describes the standard UNIX IPC facilities, pipes and signals, in terms of these features.

Section IV identifies major shortcomings of the standard UNIX IPC facilities by considering several developing applications, such as teleconferencing, network access, and user agents, that would benefit from improved IPC facilities. Most notable shortcomings are the inability to wait for multiple inputs, the inability of "unrelated" processes to communicate, and the primitive signaling facilities (see Table 2). A set of potential techniques for

improvement is also developed in the course of this discussion.

Section V elaborates these techniques, including named ports, message ports, improved signals, and message facilities that might provide desirable new capabilities. Named ports allowing communication between unrelated processes and/or message ports identifying the source of potentially multiple inputs to the port appear to have a high benefit/cost ratio. Implementation of such a facility is described in a companion report R-2064/2-AF.* Several attractive improvements to synchronization primitives have also been identified, but further research is needed to finalize an implementation plan in this area.

---

* Steven Zucker, Interprocess Communication Extensions for the UNIX Operating System: II. Implementation, The Rand Corporation, R-2064/2-AF, June 1977.

CONTENTS

I. INTRODUCTION

The UNIX operating system for the PDP-11 series of minicomputers has gained wide popularity in academic and government circles. This report considers interprocess communication (IPC) facilities with the goal of developing an improved IPC capability for the UNIX operating system [1]. A reasonable familiarity with UNIX is required to understand some detailed points, but sufficient background material is presented so that most of the discussion should stand on its own.

To see where UNIX stands in relation to other operating systems and to appreciate alternatives, existing IPC facilities were surveyed. Section II presents an outline of the major issues involved in providing IPC capabilities. The discussion focuses on functional capabilities and implementation approaches rather than on theory (i.e., developing a formalism to represent IPC). Issues are discussed at a fairly general level for brevity, although the references cited provide more detailed examples.

The remainder of this report applies to UNIX more specifically. Section III describes the current UNIX IPC capabilities in terms of the outline developed in Sec. II. Section IV explores some proposed applications (such as teleconferencing and network software) that would benefit from powerful IPC capabilities, pointing out the shortcomings of current mechanisms, and identifying additional capabilities that would be desirable. Summaries of current shortcomings and desirable additions appear at the end of Sec. IV. Section V presents several mechanisms that might be used to extend

UNIX IPC capabilities in the desired directions and begins the process of evaluating alternative techniques. A matrix showing the capabilities provided by both current and proposed mechanisms ends the section.

Assuming that a multiprocess environment is provided by the operating system (process creation, scheduling, protection), IPC facilities serve two main purposes: synchronization and data transfer. Depending on which of these functions is paramount, IPC facilities typically take different forms. Synchronization may be provided by P and V semaphores [2], BLOCK and WAKEUP [3], WAIT and SIGNAL [4,5], EVENT FLAGS [6,7], software or pseudo-interrupts [8,9]. Bulk data transmission is typically provided by shared segments [6,10], or PORTS [5,6,11,12] which allow READ and WRITE commands much like normal files. The distinction between these two applications is never complete, since synchronization primitives can be used to transmit data (Morse code at least), while the presence or absence of data can be used for synchronization purposes. Between these extremes is a class of techniques such as MAILBOXES [5,7,10,13] and MESSAGE QUEUES [6,14,15] which combine reasonably convenient data transfer and synchronization. The following section identifies the important components of these IPC techniques, although some considerations apply most clearly to a limited class of techniques.

II. ELEMENTS OF IPC

This section outlines the important considerations in designing or understanding an IPC facility. The outline presented below is by no means the only possible organization for analyzing IPC systems, and an author with different background or purpose might well produce a different one. Nevertheless, the following five areas represent one attempt to impose order on this highly multidimensional topic.

CONNECTION ESTABLISHMENT

For both data- and synchronization-oriented systems, the range of other processes that a given process can communicate with is of interest. In systems with rigid process hierarchies (e.g., tree-structured), communication is typically limited to processes with a common ancestor. In other systems, any process is a potential communication partner.

In either case, the ability to learn a partner's identity is fundamental. Certain important processes or services may have "well-known" or fixed names. To communicate with a dynamically created process, however, the name of the process must be implicitly available (e.g., to a creator process), or must be explicitly requested through some system directory facility (perhaps itself a well-known process). The name of the communication path may be an actual process name, a PORT name that must be associated with a process as part of IPC setup, a buffer address, a "communications area" address, etc.

More than one process may be allowed to produce, test, or consume on a connection. MAILBOXES are typically many-to-one, while MESSAGE QUEUES and event channels are many-to-many. PORTS may be one-to-one or many-to-many. SIGNALS may affect a particular process, an arbitrary single waiting process, or all processes waiting for a given condition. In many-to-many implementations, the identity of participants may be known (e.g., source and destination in messages) or may be invisible. Source identification serves at least three purposes: distinguishing inputs from different sources when this is desirable; providing a name for return communication, and providing a name for authorization checking. A single ID does not necessarily serve all three purposes, so multiple IDs may be desirable.

Communication paths may require explicit creation (e.g., binding ports to processes) or may be implicitly associated with processes (e.g., signals). Paths may provide only one-way (simplex) communications, two-way (duplex) communications, or simplex with reply capabilities. In the latter case, replies may return results (function call type interaction) and provide status information (acknowledgment of receipt for reliability or synchronization purposes).

Authorization for communication may be implemented by controlling the acquisition of names (i.e., the name itself serves as a capability and knowing a name guarantees ability to communicate), or names may be public with other (or no) means of access control. Whatever the entity named, the acquisition and authorization of the name must occur before communication can proceed. Authorization may occur once when a connection is explicitly opened, or on every

communication if there is no permanent connection established.

In distributed systems the identity of a partner must include a location. This location may be specified by the source, or determined "automatically" by the system (in which case the IPC supports "location independence"). In a monolithic system, all routing is done by a single switch so location is not a factor. Connection establishment in distributed systems frequently includes agreement on other mechanisms such as explicit error and flow control in addition to names. Therefore IPC with remote processes often presents a different user interface than local IPC, although some systems have taken pains to make them appear identical.

Process termination normally includes terminating all connections to the process as part of cleanup activities. Processes may also be allowed to explicitly close connections or refuse communication before their termination. This raises the questions of how to handle any data sent by or to be received by the terminated process. Other processes involved in the closed connections may be notified of the termination immediately, or later if they attempt to use the closed connection, or not at all.

RESOURCE ALLOCATION

Resource allocation for IPC is primarily a matter of buffer allocation. Whenever queues (of data, signals, or process IDs) are involved, either the source, the destination, the "owner" (e.g., of an event channel), or the "system" must provide storage space. "Flow control" concerns limitation of source and destination activity so that resources at both are not exceeded, while "congestion control"

involves equitable allocation of system resources among multiple connections to avoid hogging of storage space or processing power by any one source and destination. Buffering of each connection independently by the users eliminates the need for system resources but precludes the statistical advantage of a common (system) buffer pool. With event flags and some pseudo-interrupt systems, queuing is eliminated altogether. Since only the occurrence or non-occurrence of an event is maintained, subsequent instances of the event need not be queued.

FUNCTIONS AVAILABLE

As noted in the introduction, data-oriented IPC systems typically provide READ and WRITE commands with a count argument. Message systems provide RECEIVE and SEND commands, which perform both data transmission and synchronization. These data transmission commands may or may not block (cause the executing process to be suspended) until they are satisfied by matching communication. Blocking commands impose tighter synchronization so that communicating processes essentially become coroutines. Non-blocking commands are desirable to allow independent parallel activity by the producer(s) and consumer(s) (subject to resource limitations discussed above). However, non-blocking commands are substantially more complex because they require asynchronous (delayed) notification of status and identification of which command the reply refers to. This notification may be provided by pseudo-interrupts or signals if they are available as synchronization facilities.

Non-blocking commands require storage space to queue pending requests (see above), while blocking commands may be said to queue commands in time using the existing process suspension facilities to queue waiting processes.

Synchronization is provided by WAIT and SIGNAL commands or pseudo-interrupts. Interrupts force immediate attention, while signals must be explicitly requested by the process when it chooses (WAIT command). Of course, the WAIT command is designed to block until an appropriate SIGNAL is generated. In systems with preemption, the SIGNAL command may invoke the scheduler and result in preemption of the signaling process if the signaled process has higher priority.

Some systems provide a TEST command to indicate whether signals, data, or messages are available. Polling with the TEST command can partially overcome the limitations of blocking data transmission commands, but this requires periodic testing or "busy waiting," which consumes processing resources.

Produce, consume, and TEST commands may operate sequentially (on the next item available), randomly (on any item), by priority of available items, or selectively (allow specification of a particular type of item). The type of an item may be implicit (e.g., which of many event channels the item is stored in) or explicitly stored as a field in each item (e.g., the source process or message type).

RECEIVER UNBLOCKING

In data-oriented IPC, a RECEIVE or READ command normally returns when it is "satisfied." In stream mode, the consumer specifies the

amount of data (byte count) that must be available before the command is satisfied. In message mode, the command is satisfied at a producer designated point (the end of the message), although the receiver also specifies an upper limit (buffer size) for data transfer that satisfies the command even if a complete message has not been received.

Stream mode is appropriate when data are to be interpreted as an infinite stream of bytes and the producer has no special knowledge of logical unit size. Message mode is highly desirable when data consist of variable length logical units so the producer can initiate processing at the end of each unit. Message separation is implicit for fixed length messages, or may be implemented with a length field prefix (header), or a special end-of-message character suffix in the data stream for variable length messages.

RELATION OF IPC TO OTHER I/O

In addition to explicit IPC, processes interact with their environment via files and interrupts. Files typically correspond to data-oriented IPC, while interrupts correspond to synchronization. To unify these types of interaction with IPC, systems have attempted to make IPC "look like" files and/or interrupts, or have made files and interrupts look like IPC (e.g., messages to a file handler or "external" process, or interrupts converted to messages). As noted above, interaction with remote versus local processes in distributed systems may also be handled either uniformly, or with distinct interfaces. Unification of file, interrupt, and remote process interaction provides a great deal of flexibilty (e.g., a process can

substitute for a file) but may require additional work by the "system" in converting uniform process commands to appropriately control interaction with different elements of the environment.

III. CURRENT UNIX IPC

This section describes the current UNIX IPC facilities in terms of the general considerations developed in Sec. II. UNIX provides two facilities that directly support IPC: pipes and signals. Pipes are essentially files used for data transfer between processes, while signals (the KILL and SIGNAL commands) allow processes to cause pseudo-interrupts and to specify their handling.

CONNECTION ESTABLISHMENT

Pipes can only be used between processes with a common cooperative ancestor (FORKed processes) that sets up the pipes. Signals can only be transmitted between processes owned by the same user (normally also common descendants). No other access controls are imposed or available on IPC. The file descriptor for a pipe must be passed to created processes. The identity of processes to be signaled must be known to the signaling process (returned by FORK command). The ARPANET software (NCP, TELNET) provides for data transfer and synchronization with remote processes and potentially also with unrelated local processes, but the cost of network connections effectively precludes their use for local IPC. Theoretically, a named file could be used by arbitrary unrelated processes in the same way as a pipe, but the file might grow very long, and readers would receive an end of file (EOF) instead of blocking if they got ahead of writers. The creation and deletion of well-known file names are also used for mutual exclusion in some

cases (e.g., printer assignment).

Multiple processes may read or write on a pipe if so initialized by their creator. The identity of producers is not preserved, and data from different WRITES may be interleaved. Hence a pipe is not a reliable multiplexer of input data unless producers synchronize their activity by some other means.

Pipes (and other files) are closed when a process terminates, and may also be closed explicitly by the process. Writing a pipe with no readers causes an error, while reading a pipe with no writers returns an EOF.

RESOURCE ALLOCATION

Buffer space for pipes (and file I/O) comes from a common system buffer pool. As blocks are filled by a producer, they may be written to disk until consumed. As a means of flow and congestion control, the producers are blocked when they have written 4096 bytes until the consumers catch up, at which point written blocks are freed and the pipe is reset. (At Rand, pipes are reset whenever the consumer catches up with the producer.) For signals, resource allocation is trivial, since the last signal received is stored in a single location for each process. Any previous signals are lost (not queued).

FUNCTIONS AVAILABLE

Pipes provide simplex stream data transfer between processes. The READ command blocks until data are available (see below). The WRITE command blocks only until data are transferred into a system

buffer and there is no status returned to indicate whether the data were successfully received by the consumer. Standard UNIX provides no TEST command for pipes, but Rand has implemented the EMPTY command, which returns a Boolean value indicating whether a READ of the pipe would block (but no information is given on the amount of data in the pipe).

Pseudo-interrupts to a particular process may be caused by the KILL command. The processing of these signals may be set using the SIGNAL command to ignore, default (kill process), or a special handler routine for each type of interrupt. For most signal types, processing is reset to the default mode whenever a signal is received (but may be reinstated by an interrupt handler). The identity of the signaler is not available to the signaled process. The number of legal signals is a system generation constant. Currently about 7 signals are available for general use, while 13 have standard meanings. Since signals are not queued, there is no TEST command applicable to signals. A process cannot detect that a signal has been lost because of the occurrence of a more recent signal. An unfortunate side effect of signals is the abortion of certain types of kernel functions in progress when a signal is received. Preemption occurs if the signaled process has higher priority than the signaler.

RECEIVER UNBLOCKING

In a normal pipe (or file) READ, if some data are available, the command returns with the requested number of bytes, or all the available data, whichever is smaller. Hence the READ may be

satisfied by several smaller WRITE commands, or part of one larger WRITE. The boundary between WRITEs is not preserved, and there is no way for a writer to force completion of a READ (i.e., write an EOF). If a pipe is empty, the READ blocks until completion of the first WRITE.

If a pipe is not open for writing by any process, READ returns an EOF condition. Rand has added a command to "write" an EOF (causes EOF condition when a READ is executed) without closing the pipe. This command was added to allow a pipe to function like a teletype, which can cause zero characters (an EOF) to be "written." This pseudo-EOF could also be used to facilitate message mode operation on pipes where a large READ request would be satisfied at a producer-specified point.

RELATION OF IPC TO OTHER I/O

Pipes, device I/O, and files are purposely designed to look as much alike as possible, allowing simple substitution of one for the other in almost all cases. A few minor differences we have noted are (potential) universal accessibility of files versus the hierarchical access to pipes, the 4096 byte write-ahead limit on pipes, and the peculiar characteristics of teletype "files." Data exchange with remote processes on the ARPANET is also achieved by READ and WRITE commands to pseudo-files that have been appropriately opened (network connection establishment). Hardware interrupts (instruction traps) are converted to standard signals, allowing common handling of hardware, system, and IPC signals.

IV. SHORTCOMINGS OF CURRENT UNIX IPC

This section tries to identify the shortcomings of current UNIX IPC capabilities by considering several example applications that would benefit from improved IPC facilities. Such additional IPC facilities are outlined, while the detailed proposal of techniques to implement them is left to Sec. V. Summaries of current shortcomings and desirable additional capabilities appear at the end of this section.

TELECONFERENCING

A prototype teleconferencing system has been developed to allow simple real-time interaction between UNIX users. Basically, the input *from each participant* in the conference is copied to a conference manager process for display on other participants' terminals. Ideally, the manager should be sleeping while it awaits input from any of the participants.

This is currently difficult because: (1) if the data arrive via multiple pipes, it is only possible to READ and wait for one pipe; and (2) if the data arrive via a single pipe, the manager will wake up on any arrival but cannot determine the source of the data. A solution for (1) is to allow non-blocking READS (so multiple pipes could be read) and/or interrupts on the arrival of data in an empty pipe. (If multiple processes have the pipe open for reading should they all be interrupted?)

A solution to (2) is to create a new sort of pipe (a "message

port"), where the source of data is identified. This changes the character of pipes from stream to message, with each chunk of data written (each message) preceded by a header giving source process ID and length. For writing, a message port would be identical to a normal pipe. In reading a message port, the reader would receive the source identification along with the data. Further details of this mechanism are discussed in Sec. V.

In addition to data inputs, the conference manager also receives control commands from conference participants (e.g., show status, add participant, quit). These commands often deserve prompt attention and should not flow over the same path as normal input data. Signals carrying a small amount of data would handle this function nicely. Otherwise, commands and data would have to be mixed on a single input path, requiring some special format to differentiate them and possibly delaying commands behind data.

In general the conference manager or other processes may want to specify more complex activation conditions than the OR of several data inputs (e.g., including time events, signals from other processes or AND of conditions). This suggests supporting general Boolean expressions for activation conditions. As a minimum, improved signaling facilities are needed (e.g., avoiding loss of signals, identification of signaling process, more signals available, not aborting system functions in progress). This may require substantial system changes. The ELF system [5] provides a good example of a flexible IPC signal system. More simply, event flags for each type of signal may be adequate if multiple occurrences of the same signal are unimportant.

ARPANET SOFTWARE AND WELL-KNOWN PROCESSES

A recurring situation in network processing (e.g., TELNET, File Transfer Protocol) is the need to listen for input from either user(s) or the network. As noted above, it is not possible to have a READ pending on both the network and user inputs, resulting in various awkward ways around the fundamental dilemma. Message ports present a partial solution to this problem.

Another possibility for network processing is to make the NCP at least partially a user process. Although problematic on efficiency grounds, this allows network development and debugging to go on without impacting other users of the system. Only the fixed IMP device driver would remain in kernel code. If the NCP were a process, other processes might access it via pipes. This would require establishment of pipes between unrelated processes (e.g., user TELNET and NCP), which is not currently possible. A facility for naming pipes, which would then be called "named ports," might help to overcome this limitation. A named port essentially uses the file system to allow creation and reference to a named entity that would then be mapped to a normal (or augmented) pipe facility. Ports would provide only one-way communication, so a second port for return traffic to the user process from the NCP would be necessary. Further details are presented in Sec. V.

The assignment and distribution of port names for well-known processes (e.g., NCP, line printer process) are non-trivial problems demanding a good deal of thought. Association of ports with transitory user processes (e.g., between two shells or screen managers to "link" teletypes) is even more problematic. Typically, some directory or

globally accessible system data base is used to maintain the association of process or port IDs with well-known names (user IDs or service types). For example, the TENEX "WHERE" command returns all the processes (terminals) currently active for a given user.

RITA

The Rule-directed Interactive Transaction Agent (RITA) system [16] currently employs three processes, primarily to make additional memory space available to the system (each process is mapped to its own 64K virtual memory). The three processes essentially pass control among themselves sequentially, although there are some cases where parallel activity might be beneficial. Such concurrent activity would be facilitated if processes could be waiting for inputs from multiple sources (e.g., teletype and another RITA process). A large number of one byte pipe transfers are done to synchronize RITA process activity. A simple message facility or signals carrying data could simplify this interaction. The ability to read a specified number of bytes (a complete message) would also benefit routines that know the format of their inputs in advance.

Tables 1 and 2 summarize some shortcomings of UNIX and some desirable features not presently provided by the system.

Table 1

SUMMARY OF IPC PROBLEMS WITH CURRENT UNIX

Data Transfer (pipes):
- o Inability to wait for input from multiple sources.
- o No communication between unrelated processes.
- o No source or type identification (sometimes this is a feature, not a bug).

Synchronization (signals):
- o Some pending system (I/O) functions are aborted.
- o Signals can be lost (only most recent kept).
- o No access control (besides same user).
- o No communication between processes of different users.
- o No source identification.

Message:
- o No message facility.

Table 2

SUMMARY OF DESIRABLE IPC FUNCTIONS NOT CURRENTLY PROVIDED BY UNIX

Minimum

- o Ability to wait for multiple inputs.
- o Reliable synchronization primitives (no lost signals).

Highly Desirable

- o Ability to wait for (block until) the first of many conditions (including pipe and terminal input, timer, signals from other processes).
- o IPC to well-known but unrelated processes.
- o Identity of signaling process available to signaled process.
- o Some data passed along with signal.
- o Simple message type IPC.

Ideal?

- o Message IPC with source, length, and message type available to receiver. Selective test and receive capabilities. Messages queued (in space).
- o Boolean expressions for process activation conditions.

V. PROPOSED IPC MECHANISMS FOR UNIX

In this section we outline the implementation of techniques to provide the desirable IPC capabilities developed in Sec. IV. The mechanisms introduced in Sec. IV (ports and signals) are described in greater detail and some additional techniques are presented. A complete description of an implementation of ports may be found in a companion report [17]. We do not provide a complete evaluation of these techniques, leaving that for further discussion; but generality, ease of implementation, compatibility, and harmony with the rest of UNIX have motivated the selection of these techniques and no doubt will be important factors in their evaluation. A summary of capabilities provided by both existing and proposed UNIX IPC mechanisms appears at the end of this section in Table 3.

MESSAGE PORTS

Message ports extend the normal pipe facility by adding a header, providing source (process and/or user ID) and length information to the reader. This allows a process to receive (and wait for) input from multiple distinguishable sources on a single pipe. Data transfer also becomes more message-like than stream-like.

As noted above, writing on a message port is indistinguishable from a normal pipe, while readers must know they are dealing with a port. To read a port, a process would first read the header (fixed length) and then the remainder of the message. Alternatively, a new command might be implemented to return header information prior to or in parallel with the normal READ. If multiple readers are allowed on

a port, other processes should refrain from reading until a complete message is consumed by the current reading process. This constraint must be enforced by the "system" or by explicit cooperation between the readers to ensure the integrity of header and message structure.

Message ports are a very attractive addition because they help remedy a serious IPC deficiency at small implementation cost while preserving the feeling of UNIX. Some problems remaining are sharing a port between sources, and selective read access. For example, the conference manager described in Sec. IV may receive a long listing from one conference participant which fills the port, preventing other processes from getting their output to the manager. Even if other processes do get data into the port, the manager may want to selectively read or test for the presence of data from a particular process, rather than the first data in the port. This may be performed by a (user) routine which reads a port and sorts its contents by source (buffering in core or on disk), or may be provided by the system. The former approach may be inefficient (extra data transfers) while the latter requires substantial new system code.

INTERRUPT ON FIRST WRITE

As noted in Sec. IV, this would allow processes to wait for multiple pipes by blocking themselves until interrupted by the first WRITE on any empty pipe. Reliable operation requires improvement of the signal system so interrupts are not lost and so pending I/O is not aborted if the process is not blocked (e.g., is reading another input). It would also be convenient to pass the information on which pipe caused the interrupt to avoid polling with the EMPTY command.

NAMED PORTS

Named ports are another extension to the normal pipe facility to allow IPC between unrelated processes. As noted above, ports may be implemented as named pipes, using the UNIX file system to support naming. This again uses existing UNIX facilities (pipes and files) to maximum advantage. Ports may be based on the standard UNIX pipe facility with only naming added, or may be based on an augmented pipe facility including headers as for message ports.

Ports, like pipes, provide only one-way communication, so a pair of ports must be created and names exchanged to allow two-way communications. The creation, assignment, distribution, destruction, and dynamic management of port names are non-trivial problems. Protection and access control on ports would be provided by normal UNIX file system facilities. For example, if the NCP process were given a unique user ID, the ports it created could be protected from group and other access as desired. Although this much protection comes "for free," it is inadequate if process ID (rather than user ID) or even more sophisticated protection seems desirable.

IMPROVED SIGNALS

To avoid loss of signals, the current UNIX signal memory could be expanded from a single byte to a bit vector (one bit for each possible signal). If the source of the signal is to be delivered, then a word for each signal is required (zero if no signal, or process ID of signaler). If priority signal handling is desired, a priority list must also be maintained. These are relatively easy changes, expanding the state space of each process by 2-64 words (for

32 signals). In the typical case where there are many processes, but few pending signals, a single pointer for each process to a common pool of signal buffers might be most efficient.

Creating an event queue for each process [5,13] or even general event channels accessible to multiple processes [15] requires substantial new system code, and some form of buffer allocation and management for queue elements. Dynamic storage allocation for other system functions may already be desirable and in the works, however.

Eliminating the abortion of pending system tasks (particularly I/O) is a much more difficult problem, basically because UNIX "system" functions are performed by processes for themselves, rather than by an independent "system." Part of a process' state during system functions is saved in fixed locations (the u-structure), so that a subsequent system call from an interrupt handling routine would destroy the pending functions' state, making resumption impossible. It should be noted that in some cases, an abort is exactly what is wanted (e.g., interrupt during wait for keyboard input).

NEW "POST" MODE OF SIGNAL PROCESSING

The current UNIX system provides three modes of signal processing to the signaled process: ignore, user-supplied processing routine, and default (abort process). If the user supplies a processing routine (for each signal type of interest), certain system functions in progress when the signal arrives are aborted before entering the interrupt routines. The interrupt routine then returns to user code with an indication that the system function failed.

A new "post" mode of signal processing has been suggested where the arrival of a signal would simply be "posted" and pending system functions would not be aborted. No interrupt processing routine would be invoked, but the signal type (number) would be placed in a location for later examination by user code. Either a single location, a single location per signal type, or a stack of locations shared by all signals could be implemented. The posted signal could be kept in either the u-structure (kernel space) or a user-designated location. It would then be explicitly requested (and deleted) by a system command or normal memory access, respectively. This would provide a message type signaling facility (explicit request by receiver) in addition to the existing interrupt type facility (receiver's immediate attention forced).

Such posting of signals in its simplest form requires only a small addition to the kernel KILL processing code (e.g., an addition to the normal ignore code). The new mode would have to be encoded in the signal processing specification vector, resulting in a minor backward compatibility problem, since all possible signal processing codes already have a meaning. Passing a small amount of data along with the signal type (signaler's ID, subtype code, memory address, etc.) might prove very useful. The main advantage of this "post" signal mode is the transmission of the signal without aborting pending I/O. The need to request the signal rather than being automatically interrupted will be an advantage in some applications and a drawback in others. It may prove desirable for a process to switch a signal from post mode to interrupt mode and then block to "wait" for the signal when it has no more work to do.

SEMAPHORES

Kernel routines currently have available the SLEEP and WAKEUP commands. SLEEP blocks a process (which is executing kernel code on its own behalf) until awakened by another process with a specified 16 bit number. WAKEUP wakes up every process waiting for its 16 bit argument (an exhaustive V). Awakened processes must test to be sure the desired condition actually holds, since another awakened process may have been scheduled first. This facility is typically used by I/O routines waiting for completion of started I/O.

These semaphore commands could also be made available to user processes. However, access control would become an important consideration, since it is probably desirable to limit which processes can affect a particular semaphore and to assign unique semaphores for use by a group of processes. With more general use, efficiency would become a more serious concern, since currently all processes are searched each time a WAKEUP is performed. With further modifications, true semaphores could also be implemented.

SIMPLE MESSAGE FACILITY

A simple message facility based on the IDA system [18] has been proposed by Stockton Gaines. Each process would have a small fixed message buffer and a ready flag. The process sets the ready flag when ready to receive. Any other process may test the ready flag, or write to the process (optional blocking until receiver ready or immediate return with error if receiver not ready). Writing a message clears the ready flag (and may interrupt or wake up the receiver). The system supplies the ID of the writing process to the

receiver. Messages are queued in time (senders blocked) rather than space (no message queue). Messages are written to a particular process only. Multiple readers, message queuing, or broadcast transmission might be implemented by an intermediary process.

Since the message size is short, this mechanism is primarily aimed at synchronization rather than data transfer. Naming processes is still a difficult problem as noted above. Access control is most simply provided by the receiver discarding messages from unwanted sources (proper identification guaranteed by system). Guaranteeing equitable access among competing senders may be difficult. No message types or selective RECEIVE is provided by the system, since there is no message queue.

MORE POWERFUL MESSAGE FACILITY

A more powerful message facility with queuing (in space) of messages, selective TEST and RECEIVE on the basis of source and/or type, and small amounts of data can provide for very general process interaction [5,13]. A process is free to write its own program for message testing and reading. This type of facility largely overlaps the signal facility discussed above, with the addition of some data to each signal which may be very useful (e.g., identifying a pipe, buffer, or file name). One difference is that signals (interrupts) typically force immediate attention (unless processing is at a higher priority), while messages are queued until explicitly requested by the receiving process. Implementation cost of a general message facility is likely to be high, since there is little foundation for its attachment in the current UNIX system.

Table 3

CAPABILITIES OF CURRENT AND PROPOSED IPC MECHANISMS

Capabilities	Pipes with EMPTY	Pipes and Interrupt on First WRITE	Message Ports with EMPTY	Named Ports with EMPTY	Current Signals	Improved Signals (But Still Abort Pending I/O)	"POST" Signals	SLEEP, WAKEUP for User Processes	Simple Message (IDA)	Powerful Message (ELF)
Access allowed (global, same user, file system, common ancestor)	A	A	A	F	U	U	U	G/U	G	G
Access control (current, augmented)	C	C	A	C	C	C	C	?	?	?
Resource needs (system, user, minimal)	S	S	S	S	M	S	S/U	S	U	S
Resource allocation (good, fair, poor)	G	G	F	F	--	?	?	?	F	?
IPC type (stream, fixed message, variable message, synchronization)	St	St	V	V/St	Sy	Sy	Sy	Sy	F	F
Interrupt or post	P	P[a]	P	P	I	I	P	P	P/I	P
TEST command (selective, general, none)	G	G	G	G	N	N	S/G	N	G	S
Read selectivity (sequential, random, priority, selective)	Sq	Sq	Sq	Sq	R	P/R	?	Sl	R	Sl
Multiple producers	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Multiple consumers	Y	Y	?	?	N	N	N	N	N	N
Source identification	N	N	Y	Y/N	N	?	?	N	Y	Y
Type field	N	N	N	N	N	?	?	N	N	Y
Producer blocks	N[b]	N[b]	N[b]	N[b]	N	N	N	N	Y/N	N
Consumer blocks	Y	Y[c]	Y	Y	--	--	N	Y	N	Y
Similarity with file I/O	G	G	F	G	--	--	--	--	P	P
Similarity with interrupts	--	--	--	--	G	G	F	F	F	P
Reliability	G	F	G	G	P	F	G	F	F	G
Side effects	N	Y[d]	N	N	Y[d]	Y[d]	N	N	N	N
Generality	G	F	F	G	F	G	G	F	F	G
Compatibility	G	G	G	G	G	G	G	F	F	F
Implementation effort (high, medium, low, exists)	E	L	L	L	E	L	L	L	M	H
Known need	H	H	H	H	H	H	M	L	L	L

[a] The first WRITE to an empty pipe causes an interrupt.

[b] Producer only blocks if 4096 byte limit is exceeded.

[c] Reader blocks, but will be awakened by a WRITE to any (empty) pipe.

[d] Aborts pending system functions.

VI. CONCLUSION

Applications involving cooperating concurrent processes require powerful interprocess communication facilities. Existing UNIX IPC facilities have several weaknesses that have hampered the development of such applications. Chief among these are the inability to wait for multiple inputs, the inability of "unrelated" processes to communicate, and the primitive signaling facilities. This report outlines several techniques for improving these weaknesses including named ports, message ports, various improvements to signal facilities, and new message facilites. Named ports allowing communication between unrelated processes, and message ports identifying the source of inputs to the port appear to have a high benefit/cost ratio and have been chosen for initial implementation (see companion report [17]). Several attractive improvements to synchronization primitives have also been identified, but further research is needed to finalize an implementation plan in this area.

REFERENCES

1. Ritchie, D. M., and K. Thompson, "The UNIX Time-Sharing System," Comm. ACM 17, 7, July 1974, pp. 365-375.

2. Dijkstra, E. W., "The Structure of THE--Multiprogramming System," Comm. ACM 11, 5, May 1968, pp. 341-346.

3. Lampson, B. W., "A Scheduling Philosophy for Multiprocessing Systems," Comm. ACM 11, 5, May 1968, pp. 347-360.

4. Habermann, A. N., "Synchronization of Communicating Processes," Comm. ACM 15, 3, March 1972, pp. 171-176.

5. Retz, D. C., "Operating System Design Considerations for the Packet Switching Environment," Proc. National Computer Conf., 1975, pp. 155-160.

6. Bayer, D. L., and H. Lycklama, "MERT - A Multi-Environment Real-Time Operating System," Proc. 5th ACM Symp. Op. Sys. Principles, Austin, Texas, November 1975, pp. 33-42.

7. Wecker, S., "A Building Block Approach to Multi Function Multi Processor Operating Systems," Proc. Computer Network Systems Conf., Huntsville, Alabama, American Institute of Aeronautics and Astronautics, April 1973, New York.

8. Lampson, B., et al., "A User Machine in a Time-Sharing System," Proc. IEEE 54, 12, December 1966, pp. 1766-1774.

9. Thomas, R. H., "JSYS Traps - A TENEX Mechanism for Encapsulation of User Processes," Proc. Fall Joint Computer Conf., 1972, pp. 351-360.

10. Spier, M., and E. Organick, "The MULTICS IPC Facility," Proc. ACM 2nd Symp. Op. Sys. Principles, Princeton University, October 1969, pp. 83-91.

11. Balzer, R. M., "PORTS - A Method for Dynamic Interprogram Communication and Job Control," Proc. Spring Joint Computer Conf., 1971, pp. 485-489.

12. Akkoyunlu, E., A. Bernstein, and R. Schantz, "Interprocess Communication Facilities for Network Operating Systems," Computer, June 1974, pp. 46-55.

13. Holmgren, Steven F., et al., Illinois Inter-Process Communication Facility for UNIX, Technical Memorandum 84, Center for Advanced Computation, University of Illinois (at Urbana-Champaign), April 1977.

14. Wulf, W. A., and R. Levin (eds.), "The Message System," in The Hydra Operating System, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, June 1975, pp. 93-118.

15. Data Disc, Inc. Notes on Event Channels for UNIX, 1976.

16. Anderson, R. H., and J. J. Gillogly, Rand Intelligent Terminal Agent (RITA) Design Philosophy, The Rand Corporation, R-1809-ARPA, February 1976.

17. Zucker, Steven, Interprocess Communiation Extensions for the UNIX Operating System: II. Implementation, The Rand Corporation, R-2064/2-AF, June 1977.

18. Gaines, R. S., "An Operating System Based on the Concept of a Supervisory Computer," Comm. ACM 15, 3, March 1972, pp. 150-156.

19. Metcalfe, R. M., "Strategies for Operating Systems in Computer Networks," Proc. ACM Annual Conference, August 1972, pp. 278-281.

20. Horning, J. J., and B. Randell, "Process Structuring," ACM Computing Surveys 5, 2, March 1973, pp. 5-30.

21. Walden, D. C., "A System for Interprocess Communication in a Resource Sharing Computer Network," Comm. ACM 15, 4, April 1972, pp. 221-230.

22. Metcalfe, R. M., "Packet Communication," PhD Thesis, Harvard University, December 1973.

23. Bobrow, D. G., et al., "TENEX, A Paged Time Sharing System for the PDP-10," Comm. ACM 15, 3, March 1972, pp. 135-143.

24. Postel, J. B., Survey of Network Control Programs in the ARPA Computer Network, MITRE Tech. Report No. 6722, October 1974.

25. Brinch Hansen, P., "The Nucleus of a Multiprogramming System," Comm. ACM 13, 4, April 1970, pp. 238-241, 250.

26. Sorenson, P. G., "Interprocess Communication in Real-Time Systems," Proc. 4th ACM Symp. Op. Sys. Principles, Yorktown Heights, New York, October 1973, pp. 1-7.

27. Sevcik, K. C., et al., "Project SUE as a Learning Experience," Proc. Fall Joint Computer Conf., 1972, pp. 331-339.

28. Morenoff, E., and J. B. McLean, "Inter-Program Communications, Program String Structures, and Buffer Files," Proc. Spring Joint Computer Conf., 1967, pp. 175-183.

29. Farber, D. J., et al., "The Distributed Computing System," Proc. IEEE Computer Society International Conference, San Francisco, February 1973, pp. 31-34.

30. Holt, R. C., and M. S. Grushcow, "A Short Discussion of Interprocess Communication in the SUE/360/370 Operating System," Proc. ACM SIGPLAN-SIGOPS Interface Meeting, Savannah, Georgia, April 1973, pp. 74-78. Also SIGPLAN Notices 8, 9, Sepember 1973.

31. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," Comm. ACM 17, 10, October 1974, pp. 549-557.